Icosahedral Grid Graphs

We?

October 17, 2014

Revision History

Revision	Date	Author(s)	Description
-	01.10.2014	Carlos Osuna	Initial version

Contents

0.1 Introduction

This document shows an schema for labelling the nodes of the graph of cells, vertexes and edges of an icosahedral grid. The grid is based on a R3B2 decomposition, where each edge of the original parallelogram of the icosahedral is tri-sect, and each of the resulting triangles is bi-sect (figure **??**)

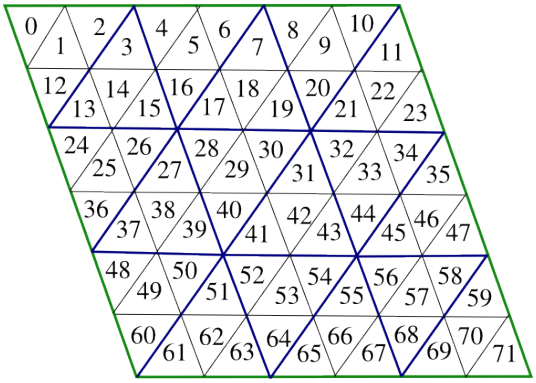

Figure 1: R3B2 decomposition of one of the parallelograms of the icosahedral. Blue lines show the first R3 decomposition, while black lines show the result of the additional B2 operation.

In this labelling, all the cells spin their upward, downward configuration in the grid, even when jumping to the next row. Least significant bit always identifies if the triangle is upward or downward. This layout of the blocks probably simplifies the looping along the cells when applying an operator. An example for the laplace follows

```
// The following code shows only modifications to Mauro's triangular_storage
// that works on the graph labelling studied here for two triangles of one of
// the icosahedral parallelograms
template<typename OffsetFunction>
struct triangular_storage {

...
  iterator begin() {
    return iterator(data.begin(), offset_function);
  }
```

```
11    iterator operator++(int) const {
12      return m_it+1;
13    }
14    iterator operator+(int i) const {
15      return iterator(m_it+i, f, toggle_direction*(i&1 ? -1 : 1));
16    }
17    double& operator[](int i) {
18      return *(m_it+f.offset(i, toggle_direction*(i&1 ? -1 : 1));
19    }
20    template <typename Functor>
21    double fold_neighbors(iterator it, Functor && f) const {
22      double v = 0;
23      for (int i=0; i<OffsetFunction::n_neighbors; ++i) {
24        v = f(v, it[i]);
25      }
26      return v;
27    }
28  };
29
30  for (int i=1; i<n-1; ++i) {
31          for (int j=1; j<m-1; ++j) {
32                  lap_cool.data[i*m+j] = 3*storage.data[i*m+j] -
33                  storage.fold_neighbors(storage.begin()+i*m+j,
34                  [](double state, double value) {
35                          return state+value;
36                  });
37          }
38  }
```

Figure also shows a possible way to tile the domain (in order to apply further parallelism) by always splitting into groups of two triangles with half size of the original parallelogram (figure **??**)

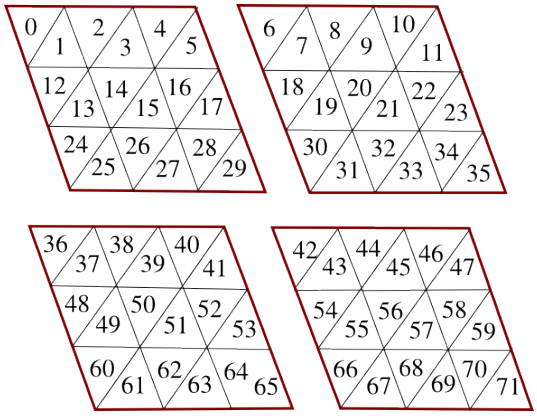

Figure 2: One possible way of tiling an icosahedral parallelogram.

0.2 Vertexes and Edges Graph

In addition to the graph of center of cells, ICON will probably need to consider the graph of vertexes (figure **??**) and edges (figure **??**). Labeling of the nodes should allow to move along these graphs in a consistent way. A

stencil operating on fields in the mass point (cell graph) might require access to the vertex graph storage or the edge graph storage. The data structure of the graph of cells could allow in the interface move into a different graph:

```
template<typename TEnv>
// structure that maps the cell graph into the vertex graph by
// providing the vertex ids around a cell id
//
// The VertexIdx is the index that identify one of the three vertexes that compose a triangle cell
// The three vertexes of a cell are numbered like (for downward triangle)
//    (0)______(1)
//     \       /
//      \     /
//       \   /
//        \ /
//        (2)
//
// or for upward triangle
//
//         (0)
//         / \
//        /   \
//       /     \
//      /_______\
//    (2)       (1)
//
//
struct cell_map {
  int m_offsets[6];
  static const int n_vertexes = 3;
  cell_map(int a) {
      m_offset[0] = 0;
      m_offset[1] = 1;
      m_offset[2] = a/2;
      m_offset[3] = 0;
      m_offset[4] = a/2+1;
      m_offset[5] = a/2;
  }
  // the following computes the vertex index of one of the vertexes
  // (determined by VertexIdx) of a cell (CellIdx)
  // Same idea should work with iterators
  int vertex_index(const int VertexIdx, const int CellIdx){
    return (CellIdx+1)/2+CellIdx/a + m_offset[VertexIdx+n_vertexes*(CellIdx&1)];
  }
};

// structure that maps the cell graph into the edges graph by
// providing the edges ids around a cell id
//
// The EdgeIdx is the index that identify one of the three edges that compose a triangle cell
// The three edges of a cell are numbered like, for downward triangle
//     ____(1)___
//     \         /
//      \       /
//      (0)   (2)
//        \   /
//         \/
//
//
// or for upward triangle
/*
//          /\
//         /  \
//       (0)  (1)
//       /      \
//      /___(2)__\
```

```
*/

struct cell_map {
  int m_offsets[6];
  static const int n_edges = 3;
  cell_map(int a) {
      m_offset[0] = 0;
      m_offset[1] = 1;
      m_offset[2] = 2;
      m_offset[3] = -1;
      m_offset[4] = 0;
      m_offset[5] = a/2*3-1;
  // the following computes the edge index of one of the edges
  // (determined by EdgeIdx) of a cell (CellIdx)
  int edge_index(const int EdgeIdx, const int CellIdx){
      return ((CellIdx+1)/2*3) + CellIdx/a + m_offset[EdgeIdx+n_edges*(CellIdx&1)];
  }
};
```

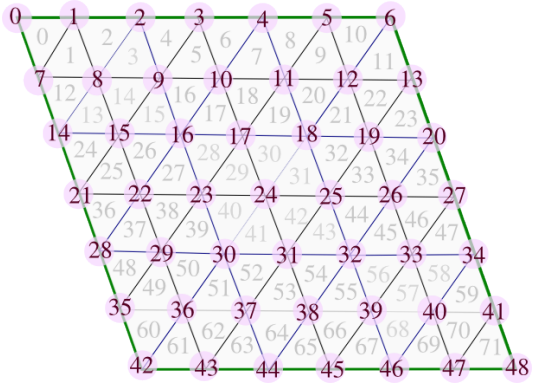

Figure 3: Graph of icosahedral vertexes.

Also we, similarly to triangular_offsets, each node of the vertex and edge graph should be able to (random) access and iterate over its neighbours. The following structure show how we can access the neighbours for the layout drawn in figure **??** and **??**:

```
//
//  indexes to refer to a vertex
//       0_______1
//      /         \
//     /           \
//    /             \
//  5       ref      2
//    \             /
//     \           /
//      \         /
//       4_______3

struct vertex_neighbour_offsets {
  int m_offsets[6];
  static const int n_neighbours = 6;
  vertex_neighbour_offsets(int a) {
```

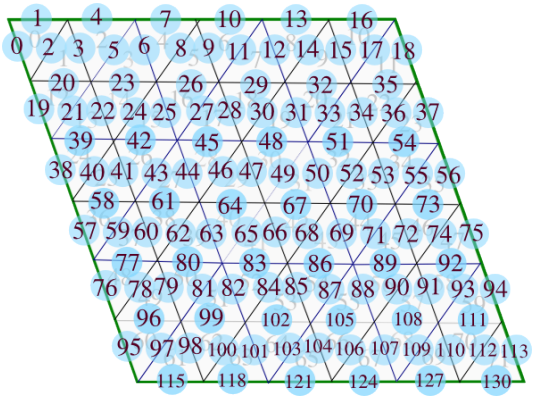

Figure 4: Graph of icosahedral edges. Note that the nodes in the last row show discontinuous index labelling in order to keep a consistent mapping between the graph of cells and the graph of edges. This requires adding some padding to the data structures.

```
    m_offset[0] = -a/2-1;
    m_offset[1] = -a/2;
    m_offset[2] = 1;
    m_offset[3] = a/2+1;
    m_offset[4] = a/2;
    m_offset[5] = -1;
  }
  // vertexes at the corners of the parallelogram will have only 5 neigoubrs
  // i.e. vertexes 0, 6, 42, 48
  // we will have to deal with these exceptional cases once we build halos around
  // the parallelogram
  int vertex_offset(int neighbour_index){
    return m_offset[neighbour_index];
  }
};
```

```
// There are three different shapes of parallelograms of edges with different
// neighbours relations. The neigbour index will is depicted below for each
// type of edge parallelogram
//
//      type I
//
//      _____1____
//     /\        /
//    /  \      /
//   0   ref   2
//  /      \  /
// /____3___\/
//
//
//     type II
//
//  _____1____
//  \        /\
//   \      /  \
//    0   ref   2
//     \  /      \
//      \/____3___\
```

```
//
//
//      type III
//
//        /\
//       /  \
//      0    1
//     /      \
//    /__ref __\
//    \        /
//     \      /
//      3    2
//       \  /
//        \/
//
struct edge_neighbour_offsets {
  int m_offsets[4];
  static const int n_neighbours = 4;
  edge_neighbour_offsets(int a) {
    m_offset[0] = -1;
    m_offset[1] = 1;
    m_offset[2] = 2;
    m_offset[3] = a/2*3-1;
    m_offset[4] = -2;
    m_offset[5] = -1;
    m_offset[6] = 1;
    m_offset[7] = a/2*3;
    m_offset[8] = -a/2*3;
    m_offset[9] = -a/2*3;
    m_offset[10] = 1;
    m_offset[11] = -1;
  }
  int edge_offset(const int neighbour_index, const int type){
    return m_offset[neighbour_index+type*n_neighbours];
  }
};
```